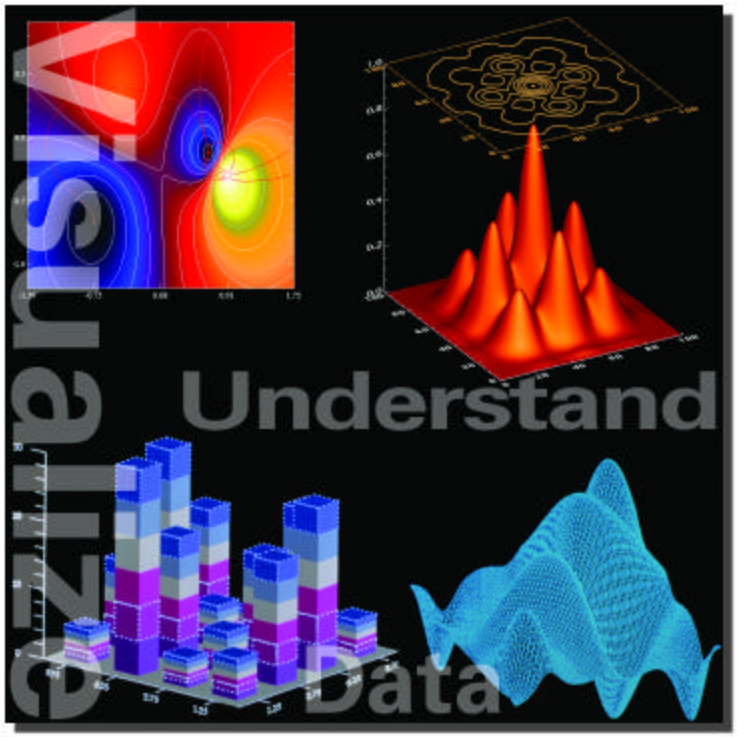
PV-WAVE 7.5®



ODBC Connection User's Guide

HELPING CUSTOMERS SOLVE COMPLEX PROBLEMS

Visual Numerics, Inc.

Visual Numerics, Inc.
2500 Wilcrest Drive
Suite 200
Houston, Texas 77042-2579
United States of America
713-784-3131
800-222-4675
(FAX) 713-781-9260
http://www.vni.com
e-mail: info@boulder.vni.com

Visual Numerics, Inc. (France) S.A.R.L.
Tour Europe
33 place des Corolles
Cedex 07
92049 PARIS LA DEFENSE
FRANCE
+33-1-46-93-94-20
(FAX) +33-1-46-93-94-39
e-mail: info@vni-paris.fr

Visual Numerics International, Ltd.
Suite 1
Centennial Court
East Hampstead Road
Bracknell, Berkshire
RG 12 1 YQ
UNITED KINGDOM
+01-344-458-700
(FAX) +01-344-458-748
e-mail: info@vniuk.co.uk

Visual Numerics, Inc.
7/F, #510, Sect. 5
Chung Hsiao E. Rd.
Taipei, Taiwan 110 ROC
+886-2-727-2255
(FAX) +886-2-727-6798
e-mail: info@vni.com.tw

Visual Numerics International GmbH
Zettachring 10
D-70567 Stuttgart
GERMANY
+49-711-13287-0
(FAX) +49-711-13287-99
e-mail: info@visual-numerics.de

Visual Numerics Japan, Inc.
Gobancho Hikari Building, 4th Floor
14 Gobancho
Chiyoda-Ku, Tokyo, 102
JAPAN
+81-3-5211-7760
(FAX) +81-3-5211-7769
e-mail: vda-sprt@vnij.co.jp

VIsual Numerics S.A. de C.V.
Cerrada de Berna 3, Tercer Piso
Col. Juarez
Mexico, D.F. C.P. 06600
Mexico

Visual Numerics, Inc., Korea
Rm. 801, Hanshin Bldg.
136-1, Mapo-dong, Mapo-gu
Seoul 121-050
Korea

Table of Contents

Preface iii

Intended Audience iii

Conventions Used in this Manual iv

Technical Support v

FAX and E-mail Inquiries vi
Electronic Services vii

Chapter 1: Getting Started 1

Introduction 1

Starting PV-WAVE ODBC Connection 2

PV-WAVE:ODBC Connection Capabilities 2

Data Source Access is Convenient 2
Use the SQL Syntax You Already Know 2
Cursor Operations are Supported 2
Multiple ODBC Connections are Supported 3
Handling Data Inside PV-WAVE 3

Supported Data Source Drivers 3

Connecting to a Data Source 4

Making a Connection to the Data Source 4
Disconnecting from the Data Source 5
Querying the Data Source 5
Example 1: Importing an Entire Table 6
Example 2: Importing and Sorting Part of a Table 7
Example 3: Importing and Sorting Table Summary Data 8
Example 4: Importing Data from Multiple Tables 9
Example 5: Importing NULL Values 11

Controlling the Rowset Size 15

Chapter 2: Reference 17

Summary of ODBC Connection Routines 17

NULL_PROCESSOR Function 19

ODBC_COMMIT Procedure 22

ODBC_CONNECT Function 23

ODBC_DISCONNECT Procedure 24

ODBC_EXIT Procedure 25

ODBC_FETCH Function 26

ODBC_INIT Function 28

ODBC_LEVEL Function 29

ODBC_META Function 30

ODBC_PREPARE Function 32

ODBC_ROLLBACK Procedure 33

ODBC_SQL Function 34

Preface

This guide explains how to use the PV-WAVE:ODBC Connection version 2.0 functions. These functions let you query an ODBC compliant data source from within PV-WAVE and import the query results into a PV-WAVE table. This imported data can then be manipulated and displayed using other PV-WAVE functions. This manual contains the following parts:

Chapter 1: Getting Started — Introduces PV-WAVE:ODBC Connection, explains how to start the module, describes what you need to know to use the software, and provides examples that demonstrate how to import data from a data source into PV-WAVE.

Chapter 2: Reference — An alphabetically arranged reference describing each of the data source connection functions.

Intended Audience

The PV-WAVE:ODBC Connection functions are easy to use if you are familiar with the target data source (such as ORACLE) and Structured Query Language (SQL). Because imported data is placed in a PV-WAVE table, you need to be familiar with the PV-WAVE table functions. These functions include BUILD_TABLE, QUERY_TABLE, and UNIQUE. They are described in the PV-WAVE Reference.

Conventions Used in this Manual

You will find the following conventions used throughout this manual:

- Code examples appear `in this typeface`. For example:

```
PLOT, temp, s02, Title = 'Air Quality'
```

- Code comments are preceded by a semicolon and are shown in this typeface, immediately below the commands they describe. For example:

```
PLOT, temp, s02, Title = 'Air Quality'
    ; This command plots air temperature data vs. sulphur dioxide
    ; concentration.
```

- Variables are shown in lowercase italics (*myvar*), function and procedure names are shown in uppercase (XYOUTS), keywords are shown in mixed case italic (*XTitle*), and system variables are shown in regular mixed case type (!Version). For better readability, all GUI development routines are shown in mixed case (WwMainMenu).
- A `$` at the end of a line of PV-WAVE code indicates that the current statement is continued on the following line. By convention, use of the continuation character (`$`) in this document reflects its syntactically correct use in PV-WAVE. This means, for instance, that *strings* are never split onto two lines without the addition of the string concatenation operator (+). For example, the following lines would produce an error if entered literally in PV-WAVE.

```
WAVE> PLOT, x, y, Title = 'Average $
   Air Temperatures by Two-Hour Periods'
       ; Note that the string is split onto two lines; an error message is
       ; displayed if you enter a string this way.
```

  The correct way to enter these lines is:

```
WAVE> PLOT, x, y, Title = 'Average '+ $
   'Air Temperatures by Two-Hour Periods'
       ; This is the correct way to split a string onto two command
       ; lines.
```

- Reserved words, such as FOR, IF, CASE, are always shown in uppercase.

Technical Support

If you have problems installing, unlocking, or running your software, contact Visual Numerics Technical Support by calling:

Office Location	Phone Number
Corporate Headquarters Houston, Texas	713-784-3131
Boulder, Colorado	303-939-8920
France	+33-1-46-93-94-20
Germany	+49-711-13287-0
Japan	+81-3-5211-7760
Korea	+82-2-3273-2633
Mexico	+52-5-514-9730
Taiwan	+886-2-727-2255
United Kingdom	+44-1-344-458-700

Users outside the U.S., France, Germany, Japan, Korea, Mexico, Taiwan, and the U.K. can contact their local agents.

Please be prepared to provide the following information when you call for consultation during Visual Numerics business hours:

- Your license number, a six-digit number that can be found on the packing slip accompanying this order. (If you are evaluating the software, just mention that you are from an evaluation site.)
- The name and version number of the product. For example, PV-WAVE 7.0.
- The type of system on which the software is being run. For example, SPARCstation, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, HP-UX 10.2 or IRIX 6.5.
- A detailed description of the problem.

FAX and E-mail Inquiries

Contact Visual Numerics Technical Support staff by sending a FAX to:

Office Location	FAX Number
Corporate Headquarters	713-781-9260
Boulder, Colorado	303-245-5301
France	+33-1-46-93-94-39
Germany	+49-711-13287-99
Japan	+81-3-5211-7769
Korea	+82-2-3273-2634
Mexico	+52-5-514-4873
Taiwan	+886-2-727-6798
United Kingdom	+44-1-344-458-748

or by sending E-mail to:

Office Location	E-mail Address
Boulder, Colorado	`support@boulder.vni.com`
France	`support@vni-paris.fr`
Germany	`support@visual-numerics.de`
Japan	`vda-sprt@vnij.co.jp`
Korea	`support@vni.co.kr`
Taiwan	`support@vni.com.tw`
United Kingdom	`support@vniuk.co.uk`

Electronic Services

Service	Address
General e-mail	`info@boulder.vni.com`
Support e-mail	`support@boulder.vni.com`
World Wide Web	`http://www.vni.com`
Anonymous FTP	`ftp.boulder.vni.com`
FTP Using URL	`ftp://ftp.boulder.vni.com/VNI/`
PV-WAVE Mailing List:	`Majordomo@boulder.vni.com`
To subscribe include:	`subscribe pv-wave YourEmailAddress`
To post messages	`pv-wave@boulder.vni.com`

CHAPTER 1

Getting Started

This chapter introduces PV-WAVE:ODBC Connection, explains how to start the module, describes what you need to know to use the software, and provides examples that demonstrate how to import data from a data source into PV-WAVE.

Introduction

PV-WAVE:ODBC Connection functions let you import data from ODBC compliant data sources into PV-WAVE. Once the data is imported, you can use PV-WAVE to analyze, manipulate, and visualize the data.

This chapter presents examples showing the following PV-WAVE:ODBC Connection routines.

- **ODBC_COMMIT** — Saves changes for an ODBC transaction.
- **ODBC_CONNECT** — Connect to an ODBC compliant data source.
- **ODBC_DISCONNECT** — Disconnect from an ODBC compliant data source.
- **ODBC_EXIT** — Exit an ODBC connection session.
- **ODBC_FETCH** — Initiates a fetch (cursor) operation.
- **ODBC_INIT** — Initiate an ODBC session.
- **ODBC_PREPARE** — Setup a cursor and prepare for fetch operations.
- **ODBC_ROLLBACK** — Cancels changes for an ODBC transaction.
- **ODBC_SQL** — Initiate an SQL command.

Starting PV-WAVE ODBC Connection

At the PV-WAVE command line, type:

```
@odbc_startup
```

This command initializes PV-WAVE:ODBC Connection.

PV-WAVE:ODBC Connection Capabilities

Data Source Access is Convenient

If you can access your data source from the computer on which PV-WAVE is running, you can connect to the data source from within PV-WAVE. This eliminates the need to export the data to a file before importing it into PV-WAVE.

Use the SQL Syntax You Already Know

You can query or update your data source from PV-WAVE using the same Structured Query Language (SQL) commands supported by your local DBMS. You do not need to learn new syntax or complicated function and procedure calls.

For database queries, PV-WAVE:ODBC Connection retrieves all of the rows and columns specified by the SQL statement. The results of the query are placed in a PV-WAVE table variable, in which each row of the query is a row in an array of type structure.

For other types of SQL statements, PV-WAVE:ODBC Connection sends the command directly to the data source. These commands can be used to insert, update, or delete data in the data source.

Cursor Operations are Supported

If your data source and ODBC driver are Level 2 compliant, you can perform read-only cursor operations on data source objects. Cursor operations allow you to retrieve part of the rows generated by a query. One application of this feature is that you can retrieve the data from a single query incrementally, processing each subset before requesting the next one. Refer to the PV-WAVE:ODBC Connection functions ODBC_PREPARE and ODBC_FETCH for more information.

Multiple ODBC Connections are Supported

You can connect to multiple ODBC drivers and/or data sources in the same PV-WAVE session, even when the data sources are administered by the same DBMS. In addition, if a particular ODBC driver and data source support multiple connections to the same data source, you can maintain multiple connections to the same data source in the same PV-WAVE session.

Handling Data Inside PV-WAVE

PV-WAVE:ODBC Connection imports query data into a PV-WAVE table variable. This format is very convenient for using other PV-WAVE procedures and functions to analyze and display the data from the query. Almost all PV-WAVE functions can access the data directly from the table. In addition, PV-WAVE has some procedures and functions that are specially designed to work with table variables. For more information, refer to the PV-WAVE functions BUILD_TABLE, QUERY_TABLE, UNIQUE, GROUP_BY, and ORDER_BY.

Supported Data Source Drivers

PV-WAVE:ODBC Connection lets you connect to many popular data source drivers, including:

- dBase
- Microsoft Access
- Excel
- FoxPro
- Paradox
- Text (comma separated values)
- Oracle
- Microsoft SQL Server

NOTE PV-WAVE:ODBC Connection can connect to any Level 1 compliant ODBC driver, although you may have to test for the actual level of functionality available based on their compliance with the ODBC standard. Fetch (cursor) operations require Level 2 compliance. Use the ODBC_LEVEL function to test for compliance level.

Connecting to a Data Source

The functions ODBC_INIT and ODBC_CONNECT are used to establish a connection to your data source from PV-WAVE. ODBC_INIT returns an ODBC environment handle, which is then used as an input argument for ODBC_CONNECT. ODBC_CONNECT returns an ODBC connection handle, which is used to identify the connection for other data source operations. The syntax is as follows:

env_handle = ODBC_INIT( )

connect_handle = ODBC_CONNECT(*env_handle*, "*data_source_name*" [, "*login_string*"])

The value of *data_source_name* must be the same as the name specified for your data source on the DSN tab of the ODBC Administrator application. If *login_string* is specified, it must contain the username and password information required for the data source. The format of this string is dependent upon the DBMS used to administer the data source. If *login_string* is not specified, then PV-WAVE:ODBC Connection uses the default values specified in ODBC Administrator.

Making a Connection to the Data Source

Assume that you would like to import some data from an Oracle data source into PV-WAVE.

First, you must load the PV-WAVE:ODBC Connection routines into your PV-WAVE session by executing the ODBC_STARTUP command file:

```
@ODBC_STARTUP
% ODBC_INITIALIZE:    PV-WAVE:ODBC Interface is initialized
```

Next, create an environment handle and make the connection. Let's assume the data source name is specified as "my_oracle_DSN" in ODBC Administrator, and that you wish to log in as user "scott", with password "tiger":

```
henv = ODBC_INIT()
hcon = ODBC_CONNECT( henv, "my_oracle_DSN", "scott/tiger")
```

You are now ready to import the data, using either ODBC_SQL, or ODBC_PREPARE and ODBC_FETCH. For more information, please refer to the PV-WAVE:ODBC Connection documentation on the ODBC_INIT and ODBC_CONNECT functions.

NOTE The ODBC driver must be installed and the data source name created in ODBC Administrator before you can access the data source from PV-WAVE. If you have problems connecting to your data source, please contact your database administrator to confirm that ODBC has been installed and configured correctly for your data source.

Disconnecting from the Data Source

Use the ODBC_DISCONNECT procedure to disconnect from the data source when you have finished interacting with it. This step can be very important if there are a limited number of license seats for connecting to the data source. Disconnecting from the data source frees a license seat so that another user can connect to the data source. If there is only one license seat available for the data source, and you wish to establish a connection to the same data source as a different user, you must end the first connection.

ODBC_DISCONNECT takes one parameter, the *connect_handle* that was returned by ODBC_CONNECT. The syntax is:

ODBC_DISCONNECT, *connect_handle*

For example, if you connected to the data source with the following ODBC_CONNECT call:

```
hcon = ODBC_CONNECT( henv, "my_oracle_DSN", "scott/tiger")
```

you can end the connection with the following ODBC_DISCONNECT call:

```
ODBC_DISCONNECT, hcon
```

This call frees the data source connection license seat for another user. For more information, please refer to the description of the ODBC_DISCONNECT procedure.

Querying the Data Source

After a connection to a data source has been established, you can use ODBC_SQL to issue any single-line SQL command to the DMBS. ODBC_SQL takes two parameters: the connection handle (returned by ODBC_CONNECT) and a string containing the SQL command. The syntax is:

result = ODBC_SQL(*connect_handle*, "*sql_command*")

The parameter *sql_command* is a string containing an SQL command to execute on the data source. If *sql_command* returns a result set (as in a SELECT statement),

result contains the result set placed in a PV-WAVE table variable. In the cases where *sql_command* does not return a result set (as in INSERT, UPDATE, or DELETE statements), *result* contains a long value that indicates the success (*result*=0) or failure (*result*=–1) status of *sql_command*. Once *result* contains a result set, *result* can be manipulated and/or displayed by any PV-WAVE routine. For instance, you can create PV-WAVE tables that are subsets of the result set (using QUERY_TABLE, for example).

NOTE PV-WAVE single-line SQL command support does not include the ability to execute Block SQL statements. Execution of stored procedures, however, is supported, so we recommend that users who wish to perform more complicated DBMS operations from PV-WAVE enclose them in a DBMS stored procedure. For more info on creating stored procedures, contact your database administrator.

Example 1: Importing an Entire Table

The ODBC_SQL command shown below imports all of the data from the table called `wave.wave_prop_trx` in an Oracle data source with the DSN `mydbserv`. The table contains 8 columns and 10000 rows.

```
henv = ODBC_INIT()
hcon = ODBC_CONNECT( henv, "mydbserv", "scott/tiger")
      ; Connect to the Oracle data source identified by DSN 'mydbserv',
      ; with username 'scott' and password 'tiger'

table = ODBC_SQL( hcon, "SELECT * FROM wave.wave_prop_trx")

INFO, table
TABLE           STRUCT    = -> TABLE_105209278474622363332715 9
   Array(10000)

INFO, table, /Structure
** Structure TABLE_105209278474622363332715 9, 8 tags, 72 length:
   TRX_ID       LONG                 0
   PROP_TYPE    STRING    'OTHER'
   PROP_ADDRESS STRING    ''
   PROP_POST_CD STRING    ''
   PROP_XGRID   DOUBLE        0.0075200000
```

```
   PROP_YGRID    DOUBLE           1.6357100
   TRX_AMT       DOUBLE           116383.00
   TRX_DATE      STRUCT    -> !DT Array(1)
```

As you can see, the data has been imported into an array of PV-WAVE structures. The tag names in the structures correspond to the column names in the database table.

Example 2: Importing and Sorting Part of a Table

In this example, we wish to import and sort a subset of the data in `wave.wave_prop_trx`. The following set of commands limits both the number of rows and columns returned to PV-WAVE.

```
henv = ODBC_INIT()
hcon = ODBC_CONNECT( henv, "mydbserv", "scott/tiger")

      ; Create the SQL command in a string PV-WAVE variable.
      ; First add the column list to this variable.
sql_command = "SELECT trx_id, prop_type, " + $
                  "trx_amt, trx_date "     + $
                  "FROM wave.wave_prop_trx "

      ; Next, add a WHERE clause to the string to limit the number of rows.
      ; The WHERE clause limits the subset to all dates between June 6, 1999
      ; and June 6, 2001
sql_command = sql_command + $
   "WHERE trx_date <= TO_DATE('2001/06/01', 'YYYY/MM/DD') " + $
   "  AND trx_date >  TO_DATE('1999/06/01', 'YYYY/MM/DD') "

      ; Finally, add an ORDER BY clause to the string to sort the dates in order.
sql_command = sql_command + "ORDER BY trx_date"
sub_table = ODBC_SQL( hcon, sql_command)


INFO, sub_table
SUB_TABLE       STRUCT    = -> TABLE_5122903921219401793313087
   Array(947)
```

```
INFO, sub_table, /Structure
** Structure TABLE_5122903921219401793313087, 4 tags, 48 length:
   TRX_ID    LONG              7514
   PROP_TYPE STRING    'OTHER'
   TRX_AMT   DOUBLE           206871.00
   TRX_DATE  STRUCT    -> !DT Array(1)


DT_TO_STR, sub_table(0).trx_date, tmp_date, tmp_time, Date_Fmt=5,
   Time_Fmt=-1


PRINT, tmp_date + " " + tmp_time
1999/06/01 22:20:37.000
```

TIP Very long SQL statements may not fit in a single PV-WAVE command string. For very long SQL statements, we recommend that you "build" the command in a PV-WAVE string variable, which can be any length.

Example 3: Importing and Sorting Table Summary Data

The ODBC_SQL command shown below imports averages by property type from table `wave.wave_prop_trx` in the Oracle data source `mydbserv`.

```
henv = ODBC_INIT()
hcon = ODBC_CONNECT( henv, "mydbserv", "scott/tiger")


amt_by_type = ODBC_SQL( hcon, "SELECT prop_type, " + $
                              "AVG(trx_amt) my_avg_amt, " + $
                              "SUM(trx_amt) my_total_amt " + $
                              "FROM wave.wave_prop_trx " + $
                              "GROUP by prop_type " + $
                              "ORDER by prop_type")
```

; Select the average transaction amount
; for each property type, ordered by property type

```
INFO, amt_by_type
AMT_BY_TYPE     STRUCT    = -> TABLE_947110189678824108225934 4
   Array(9)


INFO, amt_by_type, /Structure
** Structure TABLE_947110189678824108225934 4, 3 tags, 24 length:
   PROP_TYPE    STRING    '1BR_CONDO'
   MY_AVG_AMT   DOUBLE           80501.404
   MY_TOTAL_AMT DOUBLE           87666029.
```

NOTE When using expressions or aggregate functions in an SQL SELECT column list, we recommend that you use a column alias. This will help ensure that the tag name is valid in the PV-WAVE table variable.

This same data could also be generated with PV-WAVE functions:

```
INFO, amt_by_type_2
AMT_BY_TYPE_2   STRUCT    = -> TABLE_203312690929859568115192 2
   Array(9)


INFO, amt_by_type_2, /Structure
** Structure TABLE_203312690929859568115192 2, 3 tags, 24 length:
   PROP_TYPE    STRING    '1BR_CONDO'
   MY_AVG_AMT   DOUBLE           80501.404
   MY_TOTAL_AMT DOUBLE           87666029.
```

TIP PV-WAVE supports some searching, sorting, and aggregate functions internally (with the WHERE and QUERY_TABLE functions, for example). In many cases, the PV-WAVE searching and sorting algorithms may be faster than performing them on the DBMS server (with ODBC_SQL). We recommend that you try importing data into PV-WAVE with a minimum of sorting, and use PV-WAVE functions to sort, group, and search the data.

Example 4: Importing Data from Multiple Tables

This example combines data from three different tables into one PV-WAVE data set. The data is from air quality measurements from a number of fixed-location monitoring stations. One table contains the monitoring station location information

(wave.wave_ts_location), one contains the dataset information (wave.wave_ts_dataset), and one contains the individual measurement data (wave.wave_ts_datapoint). Notice that the tag names in the PV-WAVE table variable are the same as the column alias values given in the SELECT list.

TIP We suggest that you use explicit SELECT lists (no wildcards) and column aliases when importing data through a multi-table join.

```
henv = ODBC_INIT()
hcon = ODBC_CONNECT( henv, "mydbserv", "scott/tiger")

      ; Create the SQL command as a PV-WAVE variable
      ; This query combines data from 3 normalized tables
sql_command = "SELECT dpnt.air_temp  air_temp, " + $
       "dpnt.humidity    humidity, " + $
       "dpnt.atm_press   atm_press, " + $
       "dpnt.o3_ppm      o3_ppm, " + $
       "dpnt.co_ppm      co_ppm, " + $
       "dpnt.no2_ppm     no2_ppm, " + $
       "dpnt.pm10_ug_m3  pm10_ug_m3, " + $
       "dset.dataset_id  dataset_id, " + $
       "dset.start_date  ref_date, " + $
       "dloc.grid_x      grid_x, " + $
       "dloc.grid_y      grid_y  " + $
"FROM wave.wave_ts_datapoint dpnt, " + $
     "wave.wave_ts_dataset   dset, " + $
     "wave.wave_ts_location  dloc "

      ; Join and data limits.
      ; Only plot data for grid ID = 1
      ; And for datasets which started during 1997 through 2002.
sql_command = sql_command + $
"WHERE dset.dataset_id = dpnt.dataset_id " + $
  "AND dset.start_date >= TO_DATE('19970101', 'YYYYMMDD') " + $
  "AND dset.start_date < TO_DATE('20030101', 'YYYYMMDD') " + $
  "AND dloc.loc_id = dpnt.loc_id " + $
```

```
  "AND dloc.start_date <= dset.start_date " + $
  "AND (   dloc.end_date > dset.start_date " + $
  "     OR dloc.end_date IS NULL)  " + $
  "AND dloc.grid_id = 1 "

     ; Perform the query
table = ODBC_SQL( hcon, sql_command)
INFO, table
TABLE           STRUCT    = -> TABLE_181765074154972900728909 2
   Array(3400)

INFO, table, /Structure
** Structure TABLE_181765074154972900728909 2, 11 tags, 72 length:
   AIR_TEMP    FLOAT           29.2000
   HUMIDITY    FLOAT           26.7000
   ATM_PRESS   FLOAT           753.520
   O3_PPM      FLOAT         0.0434300
   CO_PPM      FLOAT           3.61000
   NO2_PPM     FLOAT         0.0347400
   PM10_UG_M3  FLOAT           21.1800
   DATASET_ID  LONG                 6
   REF_DATE    STRUCT    -> !DT Array(1)
   GRID_X      FLOAT          -1.46000
   GRID_Y      FLOAT           6.15000
```

NOTE PV-WAVE only supports table JOINs during data import. JOINs are not allowed on PV-WAVE table data after import.

Example 5: Importing NULL Values

PV-WAVE does not support NULL values in table variables. If PV-WAVE encounters a NULL value in a DBMS result set, it will replace it with zero (for numeric types), a NULL string (for strings), or an empty structure (for date/time values). In the following example, we use the table `wave.wave_conv_test_nulls`, which contains the following values:

```
TEST_STRING         TEST_DATE          TEST_NUM
-----------         ---------          --------
<NULL>              04-JUL-1776        3.14
<NULL_STRING>       <NULL>             0
Not null!           04-JUL-1776        <NULL>
```

In this table, <NULL> represents the database NULL value, and <NULL_STRING> is the zero-length string (‘’). The following example indicates how this table could cause problems in PV-WAVE:

```
henv = ODBC_INIT()
hcon = ODBC_CONNECT( henv, "mydbserv", "scott/tiger")
table = ODBC_SQL(hcon, "SELECT * FROM wave.wave_conv_test_nulls")

INFO, table
TABLE         STRUCT    = -> TABLE_246431244261179688304915O Array(3)

INFO, table, /Structure
** Structure TABLE_246431244261179688304915O, 3 tags, 48 length:
   TEST_STRING STRING    ''
   TEST_DATE   STRUCT    -> !DT Array(1)
   TEST_NUM    DOUBLE           3.1400000

INFO, table(1), /Structure
** Structure TABLE_246431244261179688304915O, 3 tags, 48 length:
   TEST_STRING STRING    ''
   TEST_DATE   STRUCT    -> !DT Array(1)
   TEST_NUM    DOUBLE          0.00000000

INFO, table(2), /Structure
** Structure TABLE_246431244261179688304915O, 3 tags, 48 length:
   TEST_STRING STRING    'Not null!'
   TEST_DATE   STRUCT    -> !DT Array(1)
   TEST_NUM    DOUBLE          0.00000000
```

In row 0 and row 1, the column `test_string` has the same value in PV-WAVE. However, in the database, the row 0 value is NULL and the row 1 value is the NULL string ‘’. Similarly, the values of `test_num` are the same in rows 1 and 2, even though they are different in the database.

If NULL-valued data is significant, one approach is to replace the NULL with a substitute value in the SELECT list. The following example indicates how this can be accomplished:

```
table_2 = ODBC_SQL(hcon, $
   "SELECT NVL(test_string, '_NULL_') test_string, " + $
   "NVL(test_date, TO_DATE('29991231', 'YYYYMMDD')) test_date, " + $
   "NVL(test_num,    -999999.98) test_num " + $
   "FROM wave.wave_conv_test_nulls")

INFO, table_2, /Structure
** Structure TABLE_21969278804512159187764 27, 3 tags, 48 length:
   TEST_STRING STRING    '_NULL_'
   TEST_DATE   STRUCT    -> !DT Array(1)
   TEST_NUM    DOUBLE           3.1400000

INFO, table_2(1), /Structure
** Structure TABLE_21969278804512159187764 27, 3 tags, 48 length:
   TEST_STRING STRING    ''
   TEST_DATE   STRUCT    -> !DT Array(1)
   TEST_NUM    DOUBLE          0.00000000

INFO, table_2(2), /Structure
** Structure TABLE_21969278804512159187764 27, 3 tags, 48 length:
   TEST_STRING STRING    'Not null!'
   TEST_DATE   STRUCT    -> !DT Array(1)
   TEST_NUM    DOUBLE          -999999.98
```

Another approach is the concept of *indicator variables*. An indicator variable has a value of –1 if the associated variable is NULL, and a value of zero otherwise. For an Oracle database, the following example code can be used to generate indicator variables in PV-WAVE:

```
table_3 = ODBC_SQL(hcon, $
             "SELECT test_string, " + $
              "DECODE(test_string, NULL, -1, 0) test_string_i, " + $
                 "test_date, " + $
                 "DECODE(test_date, NULL, -1, 0) test_date_i, " + $
                 "test_num, " + $
                 "DECODE(test_num, NULL, -1, 0) test_num_i " + $
             "FROM wave.wave_conv_test_nulls")

INFO, table_3, /Structure
** Structure TABLE_17757565012277460186621 68, 6 tags, 72 length:
   TEST_STRING   STRING    ''
   TEST_STRING_I DOUBLE          -1.0000000
   TEST_DATE     STRUCT    -> !DT Array(1)
   TEST_DATE_I   DOUBLE          0.00000000
   TEST_NUM      DOUBLE           3.1400000
   TEST_NUM_I    DOUBLE          0.00000000

INFO, table_3(1), /Structure
** Structure TABLE_17757565012277460186621 68, 6 tags, 72 length:
   TEST_STRING   STRING    ''
   TEST_STRING_I DOUBLE          0.00000000
   TEST_DATE     STRUCT    -> !DT Array(1)
   TEST_DATE_I   DOUBLE          -1.0000000
   TEST_NUM      DOUBLE          0.00000000
   TEST_NUM_I    DOUBLE          0.00000000
INFO, table_3(2), /Structure
** Structure TABLE_17757565012277460186621 68, 6 tags, 72 length:
   TEST_STRING   STRING    'Not null!'
   TEST_STRING_I DOUBLE          0.00000000
   TEST_DATE     STRUCT    -> !DT Array(1)
   TEST_DATE_I   DOUBLE          0.00000000
   TEST_NUM      DOUBLE          0.00000000
   TEST_NUM_I    DOUBLE          -1.0000000
```

Once the indicator variables have been created, it is a simple matter to create indices (using the WHERE function) which can be used to isolate or exclude the NULL values.

Controlling the Rowset Size

You can control the rowset size for database queries. The rowset size is defined as the number of rows that the DBMS returns to the client per network transmission.

The ability to change the rowset size allows you to tune PV-WAVE:ODBC Connection to optimize the performance of each query.

Small rowsets:

- reduce the amount of temporary memory needed to import a large dataset generated by a query.
- reduce the number of blocked processes on networks with heavy traffic.
- increase the time required to complete a query.

Large rowsets:

- reduce the time required to complete a query.
- increase the amount of temporary memory required.
- increase the number of blocked processes. .

To change the rowset size, modify the PV-WAVE system variable !Odbc_Rowset_Size. For example:

```
!Odbc_Rowset_Size = 300
```

The default value of !Odbc_Rowset_Size is 500.

All PV-WAVE:ODBC Connection routines check this system variable before accepting data from the DBMS. During the same connection, you can change the rowset size between one query and another.

For queries sent through the functions ODBC_SQL and ODBC_META, changing the rowset size does not affect the total number of rows returned, just the number of network transactions which are required to return all of the rows produced by the query. In the case of ODBC_PREPARE and ODBC_FETCH, the rowset size is the default number of rows returned for each call to ODBC_FETCH. For more details on how the value of !Odbc_Rowset_Size affects the results of ODBC_FETCH, refer to the descriptions of ODBC_PREPARE and ODBC_FETCH in the next chapter.

2

CHAPTER

Reference

Summary of ODBC Connection Routines

The syntax for these routines is summarized below:

NULL_PROCESSOR Function

table =
NULL_PROCESSOR(*null_info_object,['col1','col2',…,'coln'],Comp=comp*)

ODBC_COMMIT Procedure

ODBC_COMMIT, *connect_handle*

Saves changes for an ODBC transaction.

ODBC_CONNECT Function

connect_handle = ODBC_CONNECT(*env_handle*, *dsn* [, *login*])

Connect to an ODBC compliant data source.

ODBC_DISCONNECT Procedure

ODBC_DISCONNECT, *connect_handle*

Terminate an ODBC connection.

ODBC_EXIT Procedure

ODBC_EXIT, *env_handle*

Exit an ODBC session.

ODBC_FETCH Function

table = ODBC_FETCH(*statement_handle*, *option* [, *start_row*])

Initiates a fetch operation.

ODBC_INIT Function

env_handle = ODBC_INIT( )

Initiates an ODBC session.

ODBC_LEVEL Function

level = ODBC_LEVEL(*connect_handle*)

Tests for ODBC level compliance for specific drivers related to fetch operations only.

ODBC_META Function

table = ODBC_META(*connect_handle*, *type*, *qualifier*, *owner*, *name*)

Search for data source objects.

ODBC_PREPARE Function

statement_handle = ODBC_PREPARE(*connect_handle*, *sql_command*, *cursor_size*)

Initiate an SQL command and prepare for fetch (cursor) operations.

ODBC_ROLLBACK Procedure

ODBC_ROLLBACK, *connect_handle*

Cancels changes for an ODBC transaction.

ODBC_SQL Function

table = ODBC_SQL(*connect_handle*, *sql_stmt*)

Initiate an SQL command.

NULL_PROCESSOR Function

Facilitates the use of the *Null_Info* keyword for the DB_SQL function by extracting the list of rows containing missing for one or more columns.

Usage

table =
NULL_PROCESSOR(*null_info_object,['col1','col2',…,'coln'],Comp=comp*)

Input Parameters

null_info_object — The object returned by the *Null_Info* keyword in the DB_SQL call.

col_i — The list of column names.

Keywords

Comp=comp — Produces the complement to the result, that is, the result contains a list of rows with missing data. *comp* contains a list of rows with no missing data.

Discussion

Assuming the following use of the ODBC_SQL *Null_Info* keyword:

```
table=odbc_sql(odbc_connect('oracle', 'user_id/user_pw'), 'select *
   from blanktest', null_info=foo)
```

where `blanktest` contains the data given below, which has missing data for ID_NO in the 4^{th}, 9^{th}, and 11^{th} rows and missing data for ANIMAL_NAME in the 3^{rd}, 8^{th}, and 10^{th} rows.

ID_NO	ANIMAL_ NAME
1	golden
2	chirpy
3	NULL
NULL	harry
5	KC
6	skip
7	sparky
8	NULL
NULL	sneakers
10	NULL
NULL	harvey

NOTE Note: NULL indicates a NULL value in the corresponding database field.

Then,

```
jjj=NULL_PROCESSOR(foo,['ID_NO','ANIMAL_NAME'],Comp=comp)
```

produces the results

```
jjj = 2       3       7       8       9      10
comp = 0           1           4           5           6
```

This output can be utilized as in the following examples.

```
Table2 = table(comp)
```

produces a table with only rows and no missing values or as in the table given above.

ID_NO	ANIMAL_ NAME
1	golden
2	chirpy
5	KC
6	skip
7	sparky

Then,

```
Table3=table(jjj)
```

produces a table containing only rows with missing data (note how zeros have been substituted for values of ID_NO that are missing).

ID_NO	ANIMAL_ NAME
3	
0	harry
8	
0	sneakers
10	
0	harvey

Instead, if you want only the locations where one field is missing, a different db_sql call, jjj=foopro(foo,['ID_NO'],Comp=comp), returns an array, `jjj`, with the rows where ID_NO is missing (`3 8 10`).

Remember that rows are counted beginning with 0.

ODBC_COMMIT Procedure

Saves changes for an ODBC transaction.

Usage

ODBC_COMMIT, *connect_handle*

Input Parameters

connect_handle — A connection handle established with ODBC_CONNECT.

Keywords

Env_handle -- If set, this keyword indicates that the input parameter is an environment handle (established with ODBC_INIT). In this case, all transactions on all connections associated with this environment handle are affected.

Discussion

ODBC transactions such as UPDATE, INSERT, and DELETE can cause changes in the data on the server. Often, a user will want to review these changes and decide whether to save or cancel them. ODBC_COMMIT saves all changes which are pending on the current connection. To cancel the changes, use the ODBC_ROLLBACK procedure.

Currently, the ODBC Connection does not support savepoints or nested transactions.

NOTE Certain DBMS may allow COMMITs to occur through the ODBC_SQL command, but these are not recommended in ODBC. For best results, please use ODBC_COMMIT and ODBC_ROLLBACK.

Example

```
ODBC_COMMIT, connect_handle
```

See Also

ODBC_ROLLBACK

ODBC_CONNECT Function

Connect to an ODBC compliant data source.

Usage

connect_handle = ODBC_CONNECT(*env_handle*, *dsn* [, *login*])

Input Parameters

env_handle — An ODBC environment handle, as returned by ODBC_INIT.

dsn — String identifying a data source name (DSN).

login — A user name/password string if needed for given DSN.

Returned Value

connect_handle — An ODBC connection handle, used to identify this data source connection in other function calls. If the function fails, –1 is returned.

Discussion

A DSN (data source name) is a logical entity defining a data source under ODBC. The end user creates DSNs via the ODBC Administrator.

These entities are machine-specific rather than user specific. System DSN attributes vary from driver to driver but contain, at minimum, entries identifying the physical location of the data, the ODBC driver associated with that data, a description field, and a name field.

It is the name field that will be passed to ODBC_CONNECT. If that driver for the system DSN supports login strings, the login parameter will be used to establish the connection, overriding the default login string if established for that DSN.

Multiple connections may be open at one time. Multiple connections to the same DSN may be established if supported by the driver.

Example 1

This example shows the default connection, where the DSN is `ORACLE`.

```
oracle_id = ODBC_CONNECT(env_handle, 'ORACLE')
```

Example 2

This example shows the default connection, where the DSN is ORACLE, and the username and password are given in the login string.

```
oracle_id = ODBC_CONNECT(env_handle, 'ORACLE','scott/tiger')
```

See Also

ODBC_SQL, ODBC_DISCONNECT, ODBC_INIT

ODBC_DISCONNECT Procedure

Disconnect from an ODBC.

Usage

ODBC_DISCONNECT, *connect_handle*

Input Parameters

connect_handle — The connection handle to be freed. The connection handle is the value returned by ODBC_CONNECT.

Discussion

Use this procedure to disconnect from the data source when:

- You are finished importing data from a data source and want to end the session and free the DBMS license seat.
- You want to access the same data source, but using a different login string and the driver doesn't support multiple connections to the same DSN.

Example

In this example, the ODBC_DISCONNECT procedure is used to disconnect from the ORACLE database.

```
env_handle=ODBC_INIT()
oracle_id = ODBC_CONNECT(env_handle, 'ORACLE','scott/tiger')
emp = ODBC_SQL(oracle_id, 'SELECT * from emp')
```

```
ODBC_DISCONNECT, oracle_id
INFO, /Structure, emp
```

See Also

ODBC_SQL, ODBC_CONNECT

ODBC_EXIT Procedure

Exit the PV-WAVE:ODBC Connection environment established by ODBC_INIT.

Usage

ODBC_EXIT, *env_handle*

Input Parameters

env_handle — The environment handle returned from ODBC_INIT.

Discussion

The ODBC_EXIT call is necessary to free the ODBC environment set up by the ODBC_INIT call.

Example

```
ODBC_EXIT, env_handle
```

See Also

ODBC_INIT

ODBC_FETCH Function

Initiates a fetch (cursor) operation to retrieve a specified number of rows.

Usage

table = ODBC_FETCH(*stmt_handle*, *option* [, *start_row*])

Input Parameters

stmt_handle — The statement handle returned from ODBC_PREPARE.

option — An integer representing one of the following fetching options:

1	Fetch next — Fetch the next cursor set from the specified starting row or else the current position.
2	Fetch previous — Fetch the previous result set from the specified starting row or else the current position.
3	Fetch first — Fetch the first cursor set.
4	Fetch last — Fetch the last cursor set.
5	Next result set — If the statement produced more than one result set (a batched command or a stored procedure) go to the next result set.
6	Close — Close the statement and free its resources.

start_row — Only valid for fetch next or fetch previous options, this is the starting row for the cursor. If not specified, the fetch will go from the current position of the cursor.

Returned Value

table — Either receives a PV-WAVE table containing the results, or a count of the affected rows, or –1 for failure.

Discussion

ODBC_FETCH is used to import into a PV-WAVE table part of a DBMS server result set which was created by a call to ODBC_PREPARE. While ODBC_SQL is used to return an entire result set, ODBC_PREPARE and ODBC_FETCH allow

you to control the rate at which data is imported. Each call to ODBC_FETCH returns at most the number of rows specified by the cursor size that was determined by the call to ODBC_PREPARE.

Examples

```
orders_select = ODBC_PREPARE(dbase_orders, $
   'SELECT * FROM[orders];', 15)
```

```
orders = ODBC_FETCH(orders_select, 3)
```

orders contains rows 1-15; current position is row 16.

```
orders = ODBC_FETCH(orders_select, 1)
```

orders contains rows 16-30; current position is row 31.

```
orders = ODBC_FETCH(orders_select, 2)
```

orders contains rows 1-15; current position is row 16.

```
orders = ODBC_FETCH(orders_select, 1, 40)
```

orders contains rows 40-54; current position is row 55.

```
orders = ODBC_FETCH(orders_select, 2, 20)
```

orders contains rows 6-20; current position is row 21.

```
orders = ODBC_FETCH(orders_select, 4)
```

orders contains last rows; current position is past EOF.

```
orders = ODBC_FETCH(orders_select, 1)
```

orders is -1; current position is past EOF.

```
orders = ODBC_FETCH(orders_select, 6)
```

orders is 0.

See Also

ODBC_PREPARE

ODBC_INIT Function

Initiates an ODBC session.

Usage

env_handle = ODBC_INIT( )

Input Parameters

None.

Returned Value

env_handle — An ODBC environment handle, as returned by ODBC_INIT.

Example

```
env_handle = ODBC_INIT( )
```

This is the first call required to initialize an ODBC session.

See Also

ODBC_EXIT

ODBC_LEVEL Function

Tests for ODBC level compliance for specific drivers (related to fetch operations only).

Usage

level = ODBC_LEVEL(*connect_handle*)

Input Parameters

connect_handle — The handle returned by ODBC_CONNECT.

Returned Value

level — The level of compliance for fetch operations:

1	Indicates level 1; extended fetch operations are not possible (no cursor operations).
2	Indicates cursor operations are supported.

ODBC_META Function

Searches for data source objects.

Usage

table = ODBC_META(*connect_handle*, *option* [, *qualifier* [, *owner* [, *table_name* [, *col_name* | *table_type*]]]])

Input Parameters

connect_handle — The handle returned by ODBC_CONNECT.

option -- An integer specifying the information to retrieve, as follows:

1 Get the objects (tables, views, queries, etc.) in the data source.

2 Get the columns for one or more objects.

3 Get the permission information on the objects in the data source.

4 Get permission information on the columns of one or more objects.

qualifier — A string describing a subset of the data on the server. This could be a database, a schema, or something else, depending upon the DBMS. If this parameter is missing or a null string ('') is specified, the query returns information on all objects in the current domain. Support of SQL wildcards in this field is implementation dependent.

owner — A string describing the owner of the data on the server. If specified, only the objects or columns belonging to owners that match this parameter will be returned. If this parameter is missing or a null string ('') is specified, the query returns information on all owners. SQL wildcards (such as '%') are supported in this field.

table_name — A string describing an object name on the server. In most cases, this will be a the name of a table object, but for some DBMS, it could also be a view or a query. If specified, only the object names that match this parameter will be returned. If this parameter is missing or a null string ('') is specified, the query returns information on all objects. SQL wildcards (such as '%') are supported in this field.

col_name or ***table_type*** — A string describing a column name or table type. For *option* 1, this parameter represents the table type (TABLE, VIEW, and so on.). For *options* 2 and 4, this parameter represents the column name. For *option* 3, this parameter is ignored. If this parameter is specified, only the object information that matches this parameter will be returned. If this parameter is missing or a null string ('') is specified, the query returns information on all appropriate objects. SQL wildcards (such as '%') are supported in this field.

Returned Value

table — A PV-WAVE table containing meta-information for each object. If the function fails, –1 is returned.

Discussion

Not all drivers support all of the meta-information types. Not all drivers support searching on all of the string parameters; submit such parameters as empty strings.

Examples

```
table_info = ODBC_META( oracle_id, 1)
```

Returns a PV-WAVE table of information on all objects (tables) in the data source.

```
table_info = ODBC_META( oracle_id, 2)
```

Returns a PV-WAVE table of information on all columns of all objects (tables) in the data source.

```
table_info = ODBC_META( oracle_id, 1, '', '', 'T%')
```

Returns a PV-WAVE table of information on all objects (tables) in the data source with names that begin with T. If no such objects exist, a value of 0L is returned.

```
table_info = ODBC_META( oracle_id, 3)
```

Returns a PV-WAVE table of information on the privileges associated with all objects (tables) in the data source.

```
table_info = ODBC_META( oracle_id, 4, '', 'Bob', 'Recordings')
```

Returns a PV-WAVE table of information on the privileges associated with all columns in the table Recordings, owned by user Bob. If no such objects exist, a value of 0L is returned.

ODBC_PREPARE Function

Sets up a cursor for use with ODBC_FETCH.

Usage

stmt = ODBC_PREPARE(*connect_handle*, *sql_command*, *cursor_size*)

Input Parameters

connect_handle — The handle returned by ODBC_CONNECT.

sql_command — A string containing SQL statement(s) to execute on a data source.

cursor_size — An integer specifying the number of rows to fetch for each call to ODBC_FETCH. If this parameter is missing, *cursor_size* defaults to the value specified by the system variable !Odbc_Rowset_Size.

Returned Value

stmt — The ID for the SQL statement. This ID is used as input to the function ODBC_FETCH.

Discussion

ODBC_PREPARE is used to prepare the result set on the DBMS server, so that it can be imported into a PV-WAVE table with ODBC_FETCH.

While ODBC_SQL is used to return an entire result set, ODBC_PREPARE and ODBC_FETCH allow you to control the rate at which data is imported. Each call to ODBC_FETCH will return at most the number of rows specified by the cursor size.

Although the default cursor size is the same as the value in the system variable !Odbc_Rowset_Size, specifying the cursor size as a parameter does not change the value of !Odbc_Rowset_Size. For more information on changing the rowset size, see *Controlling the Rowset Size* on page 15.

Once a result set has been created with ODBC_PREPARE, the cursor size cannot be changed until another call is made to ODBC_PREPARE, which creates a new result set.

Example

```
orders_select = ODBC_PREPARE(dbase_orders, $
   'SELECT * FROM [orders];', 15)
```

See Also

ODBC_FETCH

ODBC_ROLLBACK Procedure

Cancels changes for an ODBC transaction.

Usage

ODBC_ROLLBACK, *connect_handle*

Input parameters

connect_handle — A connection handle established with ODBC_CONNECT.

Keywords

Env_handle — If set, this keyword indicates that the input parameter is an environment handle (established with ODBC_INIT). In this case, all transactions on all connections associated with this environment handle are affected.

Discussion

ODBC transactions such as UPDATE, INSERT, and DELETE can cause changes in the data on the server. Often, a user will want to review these changes and decide whether to save or cancel them. ODBC_ROLLBACK cancels all the changes which are pending on the current connection. To save the changes, use the ODBC_COMMIT procedure.

Currently, PV-WAVE: ODBC Connection does not support savepoints or nested transactions.

NOTE Certain DBMS may allow COMMITs to occur via the ODBC_SQL command, but these are not recommended in ODBC. For best results, please use ODBC_COMMIT and ODBC_ROLLBACK.

Example

```
ODBC_ROLLBACK, connect_handle
```

See Also

ODBC_COMMIT

ODBC_SQL Function

Initiate an SQL command.

Usage

table = ODBC_SQL(*connect_handle*, *sql_stmt*)

Input Parameters

connect_handle — The handle returned by ODBC_CONNECT.

sql_stmt — A string containing an SQL statement to execute on the data source.

Returned Value

table — A PV-WAVE table containing the result of the SQL commands, table data, or other data source response data. If the function fails or if no data is available, –1 is returned.

Keywords

Null_Info — Returns an associative array containing information on nulls in the database query result.

Discussion

For detailed information on working with tables in PV-WAVE, see the *PV-WAVE User's Guide*.

Example 1

This example imports all of the data from the `emp` table.

```
emp = ODBC_SQL(oracle_id, 'SELECT * from emp')
```

Example 2

This example imports the name, job, and salary of the managers whose salary is greater than $2800.

```
emp = ODBC_SQL(oracle_id, "SELECT ename, job," +$
   "sal from emp where job = 'MANAGER' and " + "SAL > 2800")
```

Example 3

This example imports the names and salaries of employees whose salary is between $1200 and $1400.

```
emp = ODBC_SQL(oracle_id, 'SELECT ename, sal'+$
   'from emp where sal between 1200 and 1400')
```

Example 4

This example imports the names of employees and their commissions whenever the commission is not a NULL value.

```
table = ODBC_SQL(oracle_id, 'SELECT ename' +$
   'from emp where comm is not NULL')
```

Example 5

This example uses the *Null_Info* keyword.

```
Table=odbc_sql(hcon,'select * from blanktest',null_info=foo)
```

This returns the result 'table' from your query and the null info object associative array 'foo'. Foo contains three elements:

- N_ROWS = the number of rows returned in the query
- N_COLS = the number of columns or fields returned

- MISSING_DATA = the null info object associative array

The MISSING_DATA associative array contains the field name tags, each of which has the associated array listing the rows with missing data for the tag.

For more information on the null info object and to process and extract the null information array use the NULL_PROCESSOR function.

See Also

NULL_PROCESSOR, ODBC_CONNECT, ODBC_DISCONNECT

See the following related functions in the *PV-WAVE Reference*:

BUILD_TABLE, GROUP_BY, ORDER_BY, QUERY_TABLE, UNIQUE